Problem SBH z informacją o powtórzeniach oraz wszystkimi rodzajami błędów

Bioinformatyka - sprawozdanie

Damian Tabaczyński

13.09.2020

1. WPROWADZENIE

Istnieje wiele metod sekwencjonowania DNA. Jedną z nich jest sekwencjonowanie przez hybrydyzację (w skrócie SBH). Metoda ta składa się z dwóch faz: biochemicznej i obliczeniowej. W pierwszej z nich przeprowadzany jest eksperyment hybrydyzacyjny z pełną biblioteką oligonukleotydów o pewnej ustalonej długości l. W wyniku przeprowadzenia tego eksperymentu otrzymuje się spektrum czyli zbiór wszystkich podciągów o długości l możliwych do wyróżnienia w analizowanej sekwencji DNA. W drugiej fazie - faza obliczeniowa metody sekwencjonowania przez hybrydyzację - na podstawie elementów spektrum należy zrekonstruować analizowaną sekwencję. W przypadku idealnym, proces sekwencjonowania przebiega bezproblemowo jednak w praktyce problem jest nieco bardziej skomplikowany.

W rzeczywistym eksperymencie hybrydyzacyjnym mogą wystąpić dwa rodzaje błędów: negatywne i pozytywne. Są one wynikiem niedoskonałości procesu hybrydyzacji. W przypadku błędów negatywnych spektrum nie zawiera pewnych oligonukleotydów, mimo że występują one w badanej sekwencji, natomiast błędy pozytywne oznaczają, że spektrum zawiera oligonukleotydy, których w badanej sekwencji nie ma. Ponadto do błędów negatywny zaliczamy dodatkowo błędy wynikające z powtórzeń pewnych podciągów o długości co najmniej l w analizowanej sekwencji DNA. Występowanie tych błędów wynika z ograniczeń technologicznych, które sprawiają, że w eksperymencie hybrydyzacyjnym nie można odczytać, ile razy dany fragment wystąpił w badanej cząsteczce DNA.

Rozwój technologii chipów (mikromacierzy) DNA sprawił, że staje się możliwe uzyskanie przynajmniej częściowej informacji o powtórzeniach. Informacja ta w praktyce jest nieprecyzyjna, pozwala jednak polepszyć jakość uzyskiwanych rozwiązań.

W niniejszej pracy zostanie przedstawiony przykładowy algorytm rozwiązujący problem SBH za pomocą metaheurystyki algorytmu genetycznego. Zostanie w nim użyta wcześniej wymieniona przybliżona informacja o powtórzeniach w celu poprawy jakości wyniku algorytmu. Rozważany problem został szczegółowo opisany w punkcie drugim. Natomiast w rozdziale trzecim zostanie szczegółowo opisana zasada działania algorytmu. Treść niniejszej pracy będzie również wspierana niezbędnymi przykładami i wyrywkami kodu. Przykładowa implementacja algorytmu została napisana w języku programowania *Python 3.6*.

2. OPIS PROBLEMU

Problem sekwencjonowania łańcuchów DNA na podstawie danej biblioteki oligonukleotydów o stałej długości l nad alfabetem $\{A, C, G, T\}$ tworzących łańcuch DNA o długości n przy założeniu występowania błędów pozytywnych i negatywnych oraz zawartą informacją o powtórzeniach dla każdego podanego fragmentu r poszczególnych oligonukleotydów ze spektrum. Występuje również założenie posiadania informacji o początkowym oligonukleotydzie.

Przykładowe instancje zostały wygenerowane za pośrednictwem strony dr Piotra Wawrzyniaka:

http://www.piotr.e.wawrzyniak.doctorate.put.poznan.pl/index.php/dydaktyka/bioinformatyka

3. ALGORYTM

Do rozwiązania problemu zostanie wykorzystany algorytm genetyczny, zachłanny oraz naiwny. Każdy element z podanego na wejściu spektrum będzie przydzielony do optymalnie dopasowanego ciągu oligonukleotydów w taki sposób by ciąg oligonukleotydów po nałożeniu drugiego ciągu powiększył się tylko o jedną zasadę azotową na początku lub końcu. Każdy taki ciąg optymalnie dopasowanych oligonukleotydów będzie nazywany ***łańcuchem (chain)***. Każdemu łańcuchowi zostanie dopasowany indeks za pomocą, którego będzie można go zidentyfikować lub określić jego położenie w populacji. Takie zastosowanie ułatwi późniejsze przetwarzanie na etapie operacji krzyżowania i mutacji. Każda populacja będzie składać się z ciągu łańcuchów o jak najlepszym dopasowaniu tzn. by jak najwięcej nukleotydów łańcuchów zostało nałożone na siebie (niekoniecznie będzie to optymalne). Algorytm w trakcie działania będzie eliminował gorsze populacje faworyzując te lepsze. Jednocześnie zostanie zapewnione by wyniki mogły nie wpadały wyłącznie w optimum lokalne (które nie musi być przecież optimum globalnym) przy pomocy mechanizmu mutacji i pewnych operacji krzyżowań. Algorytm podczas całego swojego działania przy łączeniu łańcuchów, krzyżowaniu lub mutacji nigdy nie pozwoli na stworzenie i dodanie do zbioru populacji większej (o dłuższym ciągu nukleotydów) niż docelowa wielkość badanej nici DNA.

Po utworzeniu początkowego zbioru populacji program sortuje strukturę malejąco i przechodzi do realizacji algorytmu genetycznego. W każdej iteracji wpierw wykonywane jest krzyżowanie populacji. Następnie przy określonych warunkach może nastąpić mutacja. Jako ostatni krok w każdym pokoleniu struktura jest sortowana. W wyniku dodawania nowych populacji przez krzyżowanie lub mutację, zbiór jest zawsze większy niż zadana na wejściu programu liczba populacji. Z tego powodu po posortowaniu struktury ***populations*** zawsze następuje selekcja ***population_size*** najlepszych populacji, które przechodzą do następnego pokolenia. Cykl ten powtarza się, aż do spełnienia warunków zakończenia. Pierwszym pokoleniem jest zbiór utworzonych populacji początkowych.

Parametry algorytmu:

```
population_size = 100  # ilość populacji w pokoleniu
max_iterations = 100  # ilość pokoleń jaka ma zostać wytworzona
crossing_frequency = 4  # współczynnik częstości krzyżowań w algorytmie
mutation_iteration_bound = 30  # po ilu rundach zacznie mutować populację
mutation_frequency = 5  # co ile rund będzie mutować
mutation_quantity = math.ceil(population_size * 0.05)  # ile zmutuje
populacji podczas jednego cyklu mutowania
n = 500  # długość docelowej pojedynczej nici DNA
l = 5  # długość pojedynczego elementu ze spektrum
```

a. Populacje początkowe

Tworzenie populacji początkowych algorytmu jest procesem wieloetapowym. Odbywa się się on z wykorzystaniem algorytmu naiwnego (losowanie) oraz zachłannego.

Początkowo brany jest cały zakres spektrum (bez pierwszego początkowego oligonukleotydu) i na jego podstawie wyznacza się wszystkie optymalne łańcuchy. Algorytm próbuje dopasować ciągi oligonukleotydów dla każdego oligonukleotydu gdy jego licznik wystąpień jest większy od 0. Jeśli program dopasował już wszystkie możliwe ciągi a istnieje jeszcze ciąg którego licznik jest większy od 0 to dodaje go jako nowy łańcuch o długości l. W celu stworzenia grupy optymalnych łańcuchów o jak największej długości wszystkie powstałe łańcuchy, o ile to możliwe, są ze sobą łączone. Każdemu elementowi z wynikowego zbioru nadawane są indeksy umożliwiające późniejszą identyfikację (oligonukleotyd początkowy jest zawsze łańcuchem o długości l i indeksie 1).

```
class Oligonucleotide:
      def __init__(self, sequence, occurrence):
      self.sequence = sequence # sekwencja nukleotydów
      self.occurrence = occurrence # ilość powtórzeń oligonukleotydu
```

```
class Chain:
      def __init__(self, sequence, new_id=-1):
      self.id = new_id # id
      self.sequence = sequence # sekwencja nukleotydów
```

Kolejnym krokiem w tworzeniu populacji początkowych jest użycie algorytmu zachłannego. W zamyśle jego działania program próbuje dodać na koniec sekwencji populacji każdy z łańcuchów z osobna. Wylicza przy tym za każdym razem ile nukleotydów może się "pokryć" lub inaczej ile nukleotydów nie będzie musiał dopisywać na koniec sekwencji populacji. W efekcie wybiera taki łańcuch, którego wynik jest największy. W przypadku łańcuchów o tym samym wyniku wybiera losowo jeden z nich. Algorytm zachłanny kończy swoje działanie w momencie wykorzystania wszystkich łańcuchów lub gdy dodanie każdego z pozostałych łańcuchów spowoduje przekroczenie limitu n. Algorytm jest uruchamiany tak długo dopóki struktura zawierająca wszystkie populację osiągnęła zadaną wartość lub odbyto już 400 iteracji.

```
class Population:
      def __init__(self):
      self.__idList = [] # lista kolejno połączonych łańcuchów
      self.points = 0 # wynik funkcji użyteczności
      self.sequence = '' # sekwencja nukleotydów
      self.length = 0 # długość sekwencji
      self.usedChainsLength = 0 # ilość wykorzystanych łańcuchów
      self.chainsLength = len(chains) # całkowita ilość łańcuchów
```

Warte zaznaczenia jest to, iż populacje są przechowywane w słowniku (mapie), gdzie kluczem jest sekwencja populacji. Dzięki temu w tej strukturze nigdy nie znajdzie się taka sama populacja. Takie zapewnienie wspomaga generowanie nowych populacji (nie wstawia duplikatów do struktury) i poprawia znacząco działanie algorytmu.

Warunek stopu dla 400 iteracji ma zapewnić, że generowanie na pewno kiedyś się zakończy - może zajść przypadek, gdzie algorytm nie wygeneruje już żadnych nowych populacji a ich całkowita liczba będzie mniejsza niż zadana. W takiej sytuacji jest uruchamiany algorytm losowy, który losowo dodaje do nowej populacji łańcuchy dopóki nie przekroczy maksymalnej długości. Algorytm naiwny (losowanie) jest uruchamiany tak długo dopóki nie zostanie osiągnięta zadana ilość populacji w pokoleniu.

b. Funkcja użyteczności

Funkcja użyteczności jest reprezentowana jedną liczbą i wyliczana jako suma dwóch wyrażeń:

- liczbę nakładających się nukleotydów pomiędzy składowymi sekwencjami oligonukleotydów
- różnice długość sekwencji wynikowej od oczekiwanej (niedomiar liczby nukleotydów od żądanej wartości).

Dla każdej populacji funkcja użyteczności jest wyliczana automatycznie po dodaniu nowego łańcucha. Wartość funkcji jest przechowywana dla każdego obiektu populacji w polu ***points***.

c. Krzyżowanie

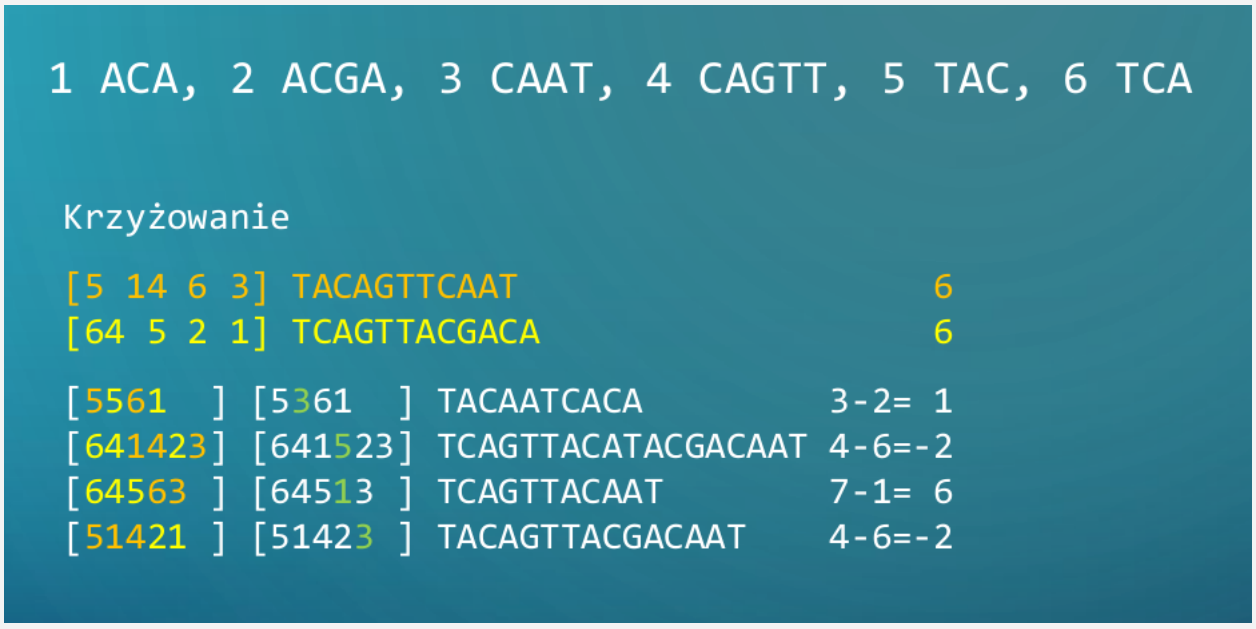
Wybór populacji do krzyżowania opiera się na promowaniu populacji o najlepszych wynikach funkcji użyteczności oraz na pseudolosowości. Z powodu losowania program uruchomiony kolejny raz może zwrócić inne wyniki. Niemniej jednak wszystkie te wyniki będą dążyć do optimum globalnego. Zapewnia to położony większy nacisk na krzyżowania lepszych populacji. Patrząc z drugiej strony krzyżowanie tylko najlepszych populacji również może wpędzić w ekstremum lokalne - co jest także niepożądane. W takim wypadku należy znaleźć złoty środek między tymi dwoma podejściami tzn. dodać możliwość krzyżowań pomiędzy populacjami najlepszymi i najgorszymi w wyznaczonej przestrzeni populacji. Zapewni to zrównoważone podejście do ekstremum globalnego przy jednoczesnym zabezpieczeniu możliwości wyjścia z lokalnego.

Algorytm wyróżnia trzy przedziały wyborów populacji:

- Krzyżowanie populacji każdy z każdym w 10% najlepszych populacjach
- W każdym przedziale wyznaczonym przez decyl następuje krzyżowanie ze sobą populacji. Liczba tych krzyżowań jest określana przez wartość zmiennej ***crossing_frequency***.
- Krzyżowanie 20% najlepszych z resztą populacji. Liczba krzyżowań z tego zakresu jest określona przez wartość wyrażenia ***8 * crossing_frequency***.

W wyniku procesu krzyżowania powstaje czwórka nowych dzieci (populacji), które mogą zostać dodane do struktury wszystkich populacji. Odbywa się to poprzez wzięcie dwóch populacji nazywanych rodzicami i wylosowaniem trzech punktów podziału. Każdy z rodziców jest dzielony na 4 części w tych samych miejscach ustalonych przez punkty podziału. Z utworzonych części buduję się kombinacje czterech nowych dzieci. Dla każdego dziecka sprawdzane jest czy w sekwencji istnieje zduplikowany łańcuch. Dla każdego zduplikowanego łańcucha wstawiamy inny nie zużyty jeszcze przez tą populację. Jeśli końcowa długość sekwencji oligonukleotydów jest dłuższa niż maksimum ***n*** to dziecko jest niepoprawne i nie dodajemy go do zbioru.

Przykład krzyżowania pokazany na ilustracji poniżej:



Operacje krzyżowania opierają się na indeksach łańcuchów co znacząco ułatwia przetwarzanie.

d. Mutacje

Mutacje przeprowadzane są u wybranych pseudolosowo populacji ze zbioru wszystkich populacji. Wyjątkiem jest tylko pierwsza najlepsza populacja, która nigdy nie zostanie poddana mutacji. Mutację nie zachodzą w każdym pokoleniu (iteracji). Ich wystąpienie jest zależne od dwóch warunków:

- przekroczenie liczby określonych pokoleń (iteracji) od której **mogą (lecz nie muszą)** rozpocząć się mutacje poszczególnych populacji. Granica ta określona jest przez zmienną ***mutation_iteration_bound.***
- ***iteration % mutation_frequency == 0*** , dane pokolenie (iteracja) musi być wielokrotnością zmiennej ***mutation_frequency.***

Ilość mutacji w każdym pokoleniu (kiedy warunki są spełnione) jest równa wartości zmiennej ***mutation_quantity.***

Dzięki tym założeniu mutacje będą następować co kilka/kilkadziesiąt pokoleń oraz dotykać tylko kilku populacji.

Mutacja polega na losowej zamianie jednego z łańcuchów (nie dotyczy początkowego) na inny, który jeszcze nie został wykorzystany przez daną populację. Warunkiem poprawnej mutacji jest by długość sekwencji populacji nie przekroczyła maksymalnego zakresu ***n***. Algorytm próbuje zamienić losowy niezużyty łańcuch z losowym łańcuchem zawierającym się w populacji. W przypadku niepowodzenia próbuje z innym łańcuchem w populacji do skutku. Jeśli to nie przyniesie oczekiwanych rezultatów wtedy ponownie próbuje mutować, lecz z innym niewykorzystanym łańcuchem. W przeważającej większość, któraś z mutacji na pewno się powiedzie. Istnieje jednak możliwość by wszystkie mutacje przebiegły niepomyślnie.

Mutacja podobnie jak krzyżowanie operuje na indeksach łańcuchów co znacząco ułatwia przetwarzanie. Należy również zaznaczyć, że dzięki mutacjom możemy zapewnić, iż rozwiązania nie wpadną w ekstremum lokalne.

e. Zakończenie algorytmu

Algorytm zakończy się po osiągnięciu ustalonego pokolenia podanego na wejściu przetwarzania. Numer tego pokolenia (maksymalna iteracja) jest przechowywana przez zmienną ***max_iterations.***

4. PRZYKŁAD ROZWIĄZAŃ

Dla instancji:

```
<dna key="25060721" length="100" start="AGGA">
<!-- [number of matches] -> [intensity signal]
  0  ->  0
  1  -> 1-3
  2  -> 3-5
  3  -> 5-7
  4  -> 6-8
  5  -> 7-8
  6  -> 7-9
  7  -> 8-9
  8  -> 8-9
  9+ ->  9
-->
    <probe pattern="NNNN">
       <cell intensity="5">AAAA</cell>
       <cell intensity="3">AAAG</cell>
       <cell intensity="3">AAAT</cell>
       <cell intensity="2">AACA</cell>
       <cell intensity="2">AAGA</cell>
       <cell intensity="1">AAGC</cell>
       <cell intensity="3">AAGG</cell>
       <cell intensity="3">AAGT</cell>
       <cell intensity="3">AATA</cell>
       <cell intensity="2">ACAT</cell>
       <cell intensity="2">ACTA</cell>
       <cell intensity="3">AGAA</cell>
       <cell intensity="3">AGAG</cell>
       <cell intensity="2">AGAT</cell>
       <cell intensity="3">AGCA</cell>
       <cell intensity="3">AGGA</cell>
```

```
        <cell intensity="2">AGGC</cell>
        <cell intensity="2">AGGT</cell>
        <cell intensity="3">AGTA</cell>
        <cell intensity="2">AGTG</cell>
        <cell intensity="3">AGTT</cell>
        <cell intensity="2">ATAC</cell>
        <cell intensity="2">ATGA</cell>
        <cell intensity="3">ATGG</cell>
        <cell intensity="2">ATTG</cell>
        <cell intensity="1">ATTT</cell>
        <cell intensity="2">CAAA</cell>
        <cell intensity="1">CAGA</cell>
        <cell intensity="2">CAGG</cell>
        <cell intensity="1">CAGT</cell>
        <cell intensity="3">CATG</cell>
        <cell intensity="3">CATT</cell>
        <cell intensity="1">CCAG</cell>
        <cell intensity="1">CCGA</cell>
        <cell intensity="3">CGAT</cell>
        <cell intensity="1">CTAA</cell>
        <cell intensity="3">CTGG</cell>
        <cell intensity="1">GAAA</cell>
        <cell intensity="3">GAAC</cell>
        <cell intensity="2">GAAG</cell>
        <cell intensity="1">GAGA</cell>
        <cell intensity="4">GAGT</cell>
        <cell intensity="2">GATC</cell>
        <cell intensity="3">GATG</cell>
        <cell intensity="2">GATT</cell>
        <cell intensity="3">GCAA</cell>
        <cell intensity="3">GCAT</cell>
        <cell intensity="2">GCCA</cell>
        <cell intensity="1">GCCG</cell>
        <cell intensity="2">GCTG</cell>
        <cell intensity="5">GGAA</cell>
        <cell intensity="1">GGAG</cell>
        <cell intensity="3">GGCC</cell>
        <cell intensity="1">GGCT</cell>
        <cell intensity="4">GGGA</cell>
        <cell intensity="3">GGGG</cell>
        <cell intensity="2">GGTT</cell>
        <cell intensity="3">GTAA</cell>
```

```
      <cell intensity="2">GTAG</cell>
      <cell intensity="2">GTGA</cell>
      <cell intensity="2">GTGC</cell>
      <cell intensity="1">GTGT</cell>
      <cell intensity="3">GTTC</cell>
      <cell intensity="5">GTTT</cell>
      <cell intensity="5">TAAG</cell>
      <cell intensity="2">TACT</cell>
      <cell intensity="1">TAGT</cell>
      <cell intensity="4">TCAG</cell>
      <cell intensity="2">TGAG</cell>
      <cell intensity="3">TGAT</cell>
      <cell intensity="2">TGCA</cell>
      <cell intensity="4">TGGC</cell>
      <cell intensity="4">TGGG</cell>
      <cell intensity="5">TGTG</cell>
      <cell intensity="3">TTCA</cell>
      <cell intensity="4">TTGG</cell>
      <cell intensity="2">TTGT</cell>
      <cell intensity="2">TTTC</cell>
      <cell intensity="4">TTTG</cell>
      <cell intensity="1">TTTT</cell>
   </probe>
</dna>
```

Lista łańcuchów:

```
-------- Chain id: 0 sequence: AGGA
-------- Chain id: 1 sequence: AAAAGAAATACTAAGCAAAA
-------- Chain id: 2 sequence: AACATGAGAGTAAGGAAC
-------- Chain id: 3 sequence: AGATGGCCAGA
-------- Chain id: 4 sequence: AGGTTCAGGCTGGC
-------- Chain id: 5 sequence: ATTTGGGAGT
-------- Chain id: 6 sequence: CAGTTTC
-------- Chain id: 7 sequence: CATT
-------- Chain id: 8 sequence: GCCGATTGGGAAGTGATC
-------- Chain id: 9 sequence: GGGG
-------- Chain id: 10 sequence: GTAGT
-------- Chain id: 11 sequence: GTGTG
-------- Chain id: 12 sequence: GTTTGTGCAT
-------- Chain id: 13 sequence: TCAG
-------- Chain id: 14 sequence: TTTT
```

Kilka przykładowych populacji początkowych:

```
-------- Points: 18 length: 99/100 chains: 14/15  sequence:
AGGAGATGGCCAGATTTGGGAGTGTGGGGTAGTTTGTGCATTTTCAGTTTCAACATGAGAGTAAGGAACAGGTTC
AGGCTGGCCGATTGGGAAGTGATC

 -------- Points: 18 length: 100/100 chains: 14/15  sequence:
AGGAGGTTCAGGCTGGCCGATTGGGAAGTGATCAGTTTCATTTGGGAGTTTGTGCATTTTGTGTGGGGTAGTAGA
TGGCCAGAACATGAGAGTAAGGAAC

 -------- Points: 6 length: 92/100 chains: 11/15  sequence:
AGGAACATGAGAGTAAGGAACATTTGGGAGTTTGTGCATTTTCAGTTTCAAAAGAAATACTAAGCAAAAGATGGC
CAGAGGTTCAGGCTGGC

 -------- Points: 13 length: 95/100 chains: 13/15  sequence:
AGGATTTGGGAGTGTGTAGTTTGTGCATTTTCAGTTTCAAAAGAAATACTAAGCAAAACATGAGAGTAAGGAACG
GGGCCGATTGGGAAGTGATC

 -------- Points: 6 length: 90/100 chains: 13/15  sequence:
AGGAGATGGCCAGATTTGGGAGTAGTGTGCCGATTGGGAAGTGATCAGTTTCATTTTGGGGTTTGTGCATAAAAG
AAATACTAAGCAAAA
```

Wyniki najlepszych 5 populacji w ostatnim 100 pokoleniu:

```
 -------- Points: 21 length: 99/100 chains: 14/15  sequence:
AGGAGGTTCAGGCTGGCCGATTGGGAAGTGATCAGTTTCATTTGGGAGTGTGTGTAGTTTGTGCATTTTAAAAGA
AATACTAAGCAAAAGATGGCCAGA    list: [0, 4, 8, 13, 6, 7, 5, 11, 11, 10, 12,
14, 1, 3]

 -------- Points: 21 length: 100/100 chains: 14/15  sequence:
AGGAGGTTCAGGCTGGCCGATTGGGAAGTGATCAGTTTCATTTGGGAGTGTGTAGTAGTTTGTGCATTTTAGATG
GCCAGAAAAGAAATACTAAGCAAAA    list: [0, 4, 8, 13, 6, 7, 5, 11, 10, 10, 12,
14, 3, 1]

 -------- Points: 21 length: 99/100 chains: 14/15  sequence:
AGGAGGTTCAGGCTGGCCGATTGGGAAGTGATCAGTTTCATTTGGGAGTAGTGTGTGTTTGTGCATTTTAGATGG
CCAGAAAAGAAATACTAAGCAAAA    list: [0, 4, 8, 13, 6, 7, 5, 10, 11, 11, 12,
14, 3, 1]

 -------- Points: 21 length: 99/100 chains: 14/15  sequence:
AGGAGGTTCAGGCTGGCCGATTGGGAAGTGATCAGTTTCATTTGGGAGTGTGTGTAGTTTGTGCATTTTAGATGG
CCAGAAAAGAAATACTAAGCAAAA    list: [0, 4, 8, 13, 6, 7, 5, 11, 11, 10, 12,
14, 3, 1]
```

```
 -------- Points: 21 length: 100/100 chains: 14/15  sequence:
AGGAGGTTCAGGCTGGCCGATTGGGAAGTGATCAGTTTCATTTGGGAGTAGTAGTGTGTTTGTGCATTTTAAAAG
AAATACTAAGCAAAAGATGGCCAGA    list: [0, 4, 8, 13, 6, 7, 5, 10, 10, 11, 12,
14, 1, 3]
```

5. BIBLIOGRAFIA

- SEKWENCJONOWANIE DNA Z BŁĘDAMI NEGATYWNYMI ORAZ INFORMACJĄ O POWTÓRZENIACH 2008, K. Kwarciak, M. Radom, R. Formanowicz http://delibra.bg.polsl.pl/Content/46717/BCPS_51106_2008_Sekwencjonowanie-DNA.pdf
- http://www.piotr.e.wawrzyniak.doctorate.put.poznan.pl/index.php/dydaktyka/bioinformatyka
- http://www.piotr.e.wawrzyniak.doctorate.put.poznan.pl/images/prezentacje/Bioinformatyka-SBH-algorytmy-heurystyczne.pdf
- https://www.wallpaperflare.com/3-d-abstraction-dna-genetic-molecule-pattern-psychedelic-wallpaper-bbybb